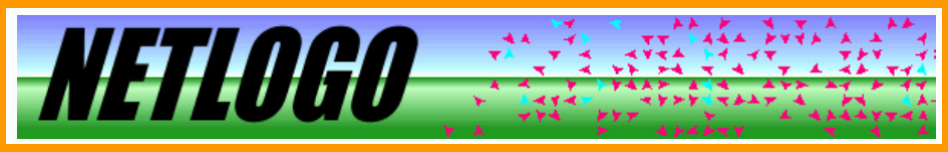


di

Stefano Cacciaguerra

Intelligenza Artificiale
AA 2006-2007

Netlogo!

- **NetLogo** is a programmable modeling environment for simulating complex systems.
- Modelers can give instructions to hundreds or thousands of independent "**agents**" all operating in parallel.
- This makes it possible to explore the connection between:
  - the **micro-level** behavior of individuals
  - the **macro-level** patterns that emerge from the interaction of many individuals.

http://www.ccl.sesp.northwestern.edu/netlogo/

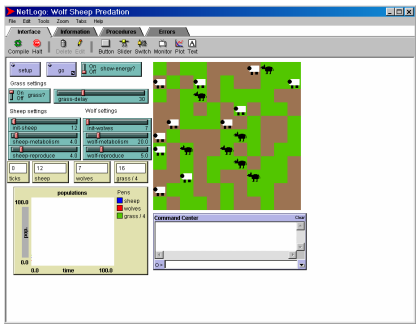
Last Version 3.1.3

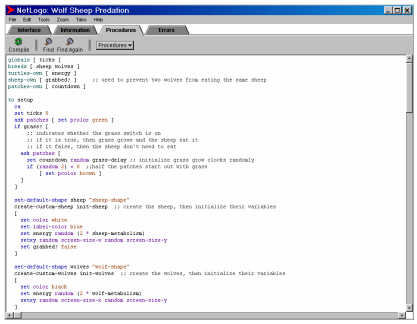
Intelligenza Artificiale
AA 2006-2007

Features

You can use the list below to help familiarize yourself with the features NetLogo has to offer.

- **System**
- **Language**
- **Environment**





Features (System)

System:

- Cross-platform: runs on MacOS, Windows, Linux, et al
- Web-enabled (run within a web browser or download and run locally)
- Models can be saved as applets to be embedded in web pages

Features(Language)

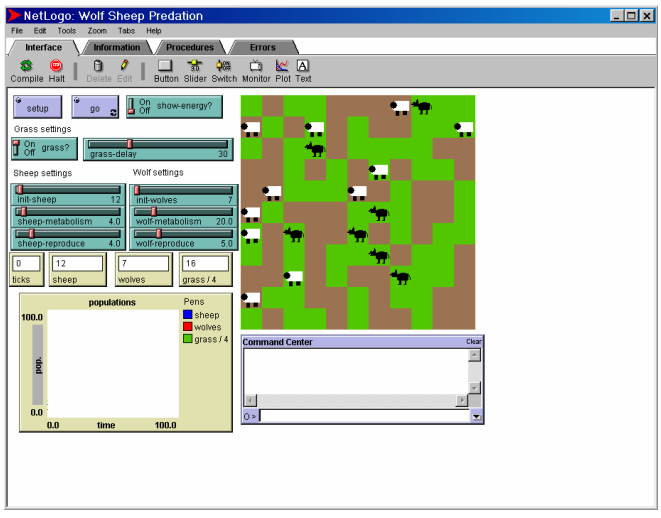
Language:

- Fully programmable
- Simple language structure
- Language is Logo dialect extended to support agents and parallelism
- Unlimited numbers of agents and variables
- Double precision arithmetic
- Many built-in primitives

Features (Environment)

Environment:

- Interface builder w/
- buttons,
- sliders,
- monitors,
- switches,
- Plots,
- text boxes.



What is Netlogo?

- Info area for annotating your model
- Powerful and flexible plotting system
- HubNet: participatory simulations using networked devices (including handhelds)
- Agent Monitors for inspecting agents
- BehaviorSpace: a tool used to collect data from multiple runs of a model
- Export and import functions (export data, save and restore state of model)
- Converts StarLogoT models into NetLogo models

Programming Guide

The following material explains some important features of programming in NetLogo.

- Agents
- Procedures
- Variables
- Colors
- Ask
- Agentsets
- Breeds
- Synchronization
- Procedures (advanced)
- Lists
- Strings
- Turtle Shapes

Agents

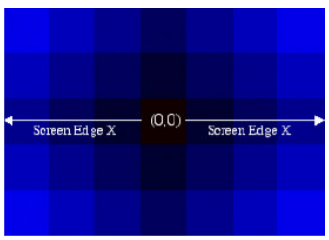
The NetLogo world is made up of agents. Agents are beings that can follow instructions. Each agent can carry out its own activity, **all simultaneously**.

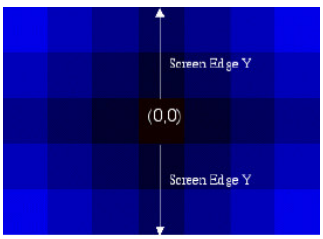
In NetLogo, there are three types of agents:

- **Turtles** are agents that move around in the world. The world is two dimensional and is divided up into a grid of patches.
- Each **patch** is a square piece of "ground" over which turtles can move.
- The **observer** doesn't have a location -- you can imagine it as looking out over the world of turtles and patches.

Patches

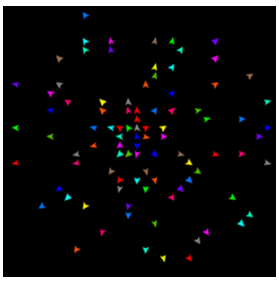
- Patches have coordinates. The patch in the center of the world has coordinates (0, 0). We call the patch's coordinates **pxcor** and **pycor** : integers.
- the standard mathematical coordinate plane
- The total number of patches is determined by the settings **screen-edge-x** and **screen-edge-y**.

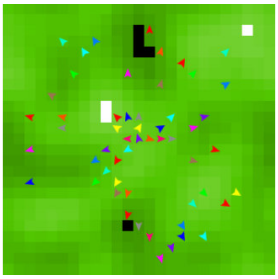




Turtles

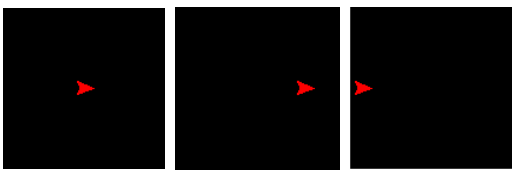
- Turtles have coordinates too: **xcor** and **ycor**.
- The each turtle has an identificator **who**.
- For speed, NetLogo always draws a turtle on-screen as if it were standing in the center of its patch, but in fact, the turtle can be positioned at any point within the patch.





Miscellaneous

- The world of patches isn't bounded, but "**wraps**" so when a turtle moves past the edge of the world, it disappears and reappears on the opposite edge.



- Every patch has the same number of "**neighbor**" patches. If you're a patch on the edge of the world, some of your "neighbors" are on the opposite edge.

Primitives

Commands and reporters tell agents what to do:

- **Commands** are actions for the agents to carry out.
- **Reporters** carry out some operation and report a result either to a command or another reporter.

Commands and reporters built into NetLogo are called **Primitives**:

Alphabetical: A B C D E F G H I J L M N O P R S T U V W X Y
Categories: Turtle Patch Agentset Color Control/Logic Display HubNet I/O List String Math Plotting
Special: Variables Keywords Constants

Procedures

- Commands and reporters you define yourself are called **procedures**.
- Each procedure has a name, preceded by the keyword **to**. The keyword **end** marks the end of the commands in the procedure.
- Once you define a procedure, you can use it **elsewhere** in your program.
- Many commands and reporters take **inputs** values that the command or reporter uses in carrying out its actions.

Examples: procedures

```
to setup
    ca ;; clear the screen
    crt 10 ;; create 10 new turtles
end

to go
    ask turtles
    [ fd 1 ;; all turtles move forward one step
    rt random 10 ;; ...and turn a random amount
    lt random 10 ]
end
```

Variables

A variable can be :

- If a variable is a **global variable**, there is only one value for the variable, and every agent can access it.
- Each turtle has its own value for every **turtle variable**, and each patch has its own value for every **patch variable**.
- Some variables are built into NetLogo: all turtles have a color variable, and all patches have a pcolor variable.(The patch variable begins with "**p**")

Variables

- You can make a global variable by adding a **switch** or a **slider** to your model, or by using the **globals** keyword at the beginning of your code:

```
globals [ clock ]
```

- You can also define new turtle and patch variables using the **turtles-own** and **patches-own**:

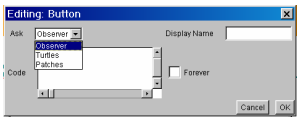
```
turtles-own [ energy speed ]
patches-own [ friction ]
```

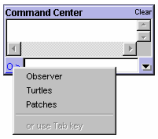
- Use the set command to set them (default value is zero).
- Global variables can by read and set at **any time** by any agent. As well, a turtle can read and set patch variables of the patch it is **standing on**.

Ask

Ask specifies commands that are to be run by **turtles** or **patches**.

- All code to be run by turtles **must** be located in a turtle "**context**":
  - In a **button**, by choosing "Turtles" from the popup menu.
  - In the **Command Center**, by choosing "Turtles" from the popup menu.
  - By using **ask** turtles.





- The same goes for patches and the observer, except that code to be run by the observer must not be inside any ask.

Examples: ask

```
to setup
    ca
    crt 100 ;; create 100 turtles
    ask turtles
    [
        set color red ;; turn them red
        rt random 360 ;; give them random headings
        fd 50 ;; spread them around
    ]
    ask patches
    [
        if (pxcor > 0) ;; patches on the right side
        [set pcolor green ] ;; of the screen turn green
    ]
end
```


Ask

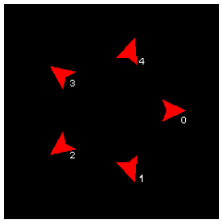
You can also use ask to make an **individual** turtle or patch run commands. The reporters **turtle**, **patch**, and **patch-at** are useful for this technique:

```
to setup
    ca
    crt 3 ;; make 3 turtles
    ask turtle 0 ;; tell the first one...
        [ fd 1 ] ;; ...to go forward
    ask turtle 1 ;; tell the second one...
        [ set color green ] ;; ...to become green
    ask patch 2 -2 ;; ask the patch at coords (2,-2)
        [ set pcolor blue ] ;; ...to become blue
    ask turtle 0 ;; ask the first turtle
        [ ask patch-at 1 0 ;; ...to ask patch to the east
            [ set pcolor red ]] ;; ...to become red
end
```

Ask

- Every **turtle** created has an **ID number**. The first turtle created has ID 0, the second turtle ID 1, and so forth.

  The turtle primitive reporter takes an ID number as an input, and reports the turtle with that ID number.

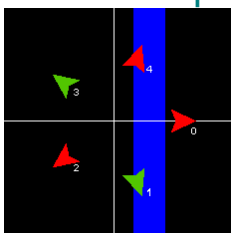


- The **patch** primitive reporter takes values for **pxcor** and **pycor** and reports the patch with those coordinates.
- The **patch-at** primitive reporter takes *offsets*: distances, in the x and y directions, *from* the first agent.

Agentsets

An **agentset** is a set of agents and it can contain either turtles or patches, but not both at once.

- You can construct agentsets that contain only **some** turtles or **some** patches. For example, all the red turtles, or the patches with pxcor equal one.



- These agentsets can then be used by ask or by various reporters that take agentsets as inputs.

Agentsets

Using **turtles-here** or **turtles-at** to make an agentset containing only the turtles on my patch, or only the turtles on some other particular patch.

- turtles

```
turtles with [color = red] ;; all red turtles
turtles-here with [color = red] ;; all red turtles on my patch
turtles in-radius 3 ;; all turtles less than 3 patches away
```

- patches

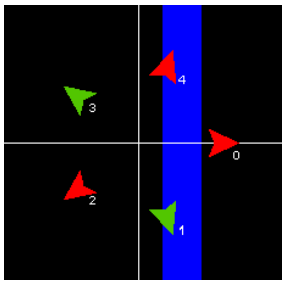
```
patches with [pxcor > 0] ;; patches on right side of screen
;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
Neighbors ;; shorthand for those eight patches
```


Agentsets

- Here are some simple things you can do:

  Use **ask** to make the agents in the agentset do something
  Use **any** to see if the agentset is empty
  Use **count** to find out how many agents are in the set

- Here are some more complex things you can do:
  random-one-of, sprout, max-one-of or min-one-of
  histogram, values-from



Breeds

You can define different "**breeds**" of turtles. The different breeds can behave differently.

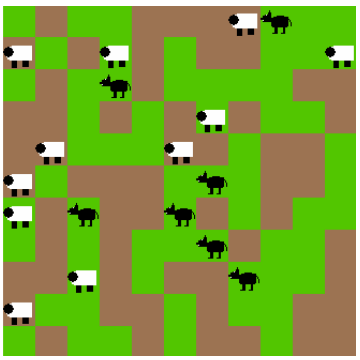
- You define breeds using the breeds keyword, at the top of your model, before any procedures: breeds [wolves sheep]
- When you define a breed such as sheep, an agentset for that breed is automatically created, so that all of the agentset capabilities described above are immediately available with the sheep agentset.
- The following new primitives are also automatically available once you define a breed:

  create-sheep, create-custom-sheep, sheep-here, and sheep-at.
- Also, you can use sheep-own to define new turtle variables that only turtles of the given breed have.

Breeds

- A turtle's breed agentset is stored in the **breed turtle variable**. So you can test a turtle's breed, like this:
  if breed = wolves [...]
- Note also that turtles can change breeds. A wolf doesn't have to remain a wolf its whole life.
  ask random-one-of wolves [set breed sheep]

```
breeds [sheep wolves]
sheep-own [grass]
to setup
    ca
    create-custom-sheep 50
    [ set color white]
    create-custom-wolves 50
    [ set color black ]
end
```



Synchronization

Turtle commands are executed **asynchronously**; each turtle does its list of commands as fast as it can.

One could make the turtles "**line up**" by waiting the end of an ask block. At that point, the turtles would wait until all were finished before any went on.

- the two steps are **not synced**:
  ask turtles
  [fd random 10
  do-calculation]

- the two steps are **synced** :
  ask turtles [fd random 10]
  ask turtles [do-calculation]

Procedures with inputs

Your own procedures can take **inputs**, just like primitives do.

To create a procedure that accepts inputs, include a list of input names in square brackets after the procedure name:

```
to draw-polygon [num-sides size]
    pd
    repeat num-sides
          [ fd size
          rt (360 / num-sides) ]
end
```

Reporter procedures

You can define your own reporters. You must do two special things:

- First, use **to-report** instead of **to** to begin your procedure.
- Second, in the body of the procedure, use **report** to report the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
  [ report number ]
  [ report 0 - number ]
end
```


Procedures with local variables

- A local variables is defined and used only in the context of a particular procedure.

- To add a local variable to your procedure, use the **locals** keyword. It must come at the beginning of your procedure:

```
to swap-colors [turtle1 turtle2]
  locals [temp]
  set temp color-of turtle1
  set (color-of turtle1) (color-of turtle2)
  set (color-of turtle2) temp
end
```

Lists

- In the simplest models, each variable holds only one piece of information, usually a **number** or a **string**.
- The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of information in a list.
- Each value in the list can be **any type** of value: a number, or a string, an agent or agentset, or even another list.
- Lists allow for the convenient packaging of information in NetLogo. If your agents carry out a repetitive calculation on multiple variables, it might be easier to have a list variable, instead of multiple number variables.

Constant Lists

- You can make a list by simply putting the values you want in the list between **brackets**, like this:

```
set mylist [2 4 6 8]
```

  Note that the individual values are separated by spaces.

- You can make lists that contains numbers and strings this way, as well as lists within lists:

```
set mylist [[2 4] [3 5]]
```

- The empty list is written by putting nothing between the brackets, like this: [].

Building Lists on the Fly

The **list** reporter accepts two other reporters, runs them, and reports the results as a list.

- If you wanted a list to contain two random values, you might use the following code:

  set random-list list (random 10) (random 20)

  This will set random-list to a new list of two random numbers each time it runs.

- To make longer lists, use the list reporter with the sentence reporter, which concatenates two lists (combines their contents into a single, larger list).

Building Lists on the Fly

- The **values-from** primitive lets you construct a list from an agentset.
- It reports a list containing the each agent's value for the given reporter.
- The reporter could be a simple variable name, or a more complex expression -- even a call to a procedure defined using to-report.
- A common idiom is

  max values-from turtles [...]

  sum values-from turtles [...]

Changing List Items

- The replace-item replace index element in the list with new value. (0 means the first item, 1 means the second item, and so forth)
- **replace-item** index list value

  set mylist [2 7 5 B [3 0 -2]] ; mylist is now [2 7 5 B [3 0 -2]]

  set mylist replace-item 2 mylist 10
  ; mylist is now [2 7 10 B [3 0 -2]]
- To add an item, say 42, to the end of a list, use the **lput** reporter. (**fput** adds an item to the beginning of a list.)

  set mylist lput 42 mylist ; mylist is now [2 7 10 B [3 0 -2] 42]

Changing List Items

- The **but-last** reporter reports all the list items but the last.

  set mylist but-last mylist
  ; mylist is now [2 7 10 B [3 0 -2]]
- Suppose you want to get rid of item 0, the 2 at the beginning of the list (but-first).

  set mylist but-first mylist
  ; mylist is now [7 10 B [3 0 -2]]

Strings

- To input a constant string, surround it with double quotes(The empty string is written like this: "").
- Most of the list primitives work on strings as well:

butfirst "string" => "tring"
butlast "string" => "strin"
empty? "" => true
empty? "string" => false
first "string" => "s"
item 2 "string" => "r"
last "string" => "g"
length "string" => 6

Strings

member? "s" "string" => true
member? "rin" "string" => true
member? "ron" "string" => false
position "s" "string" => 0
position "rin" "string" => 2
position "ron" "string" => false
remove "r" "string" => "sting"
remove "s" "strings" => "tring"
replace-item 3 "string" "o" => "strong"
reverse "string" => "gnirts"

Strings

- Strings can be compared using the **=, !=, <, >, <=,** and **>=** operators.
- To concatenate strings, that is, combine them into a single string, you may also use the + (plus) operator:
- "tur" + "tle" => "turtle“
- If you need to embed a special character in a string, use the following escape sequences:

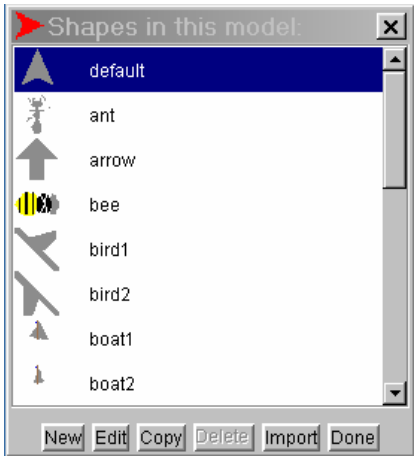
  \n = newline (carriage return)
  \t = tab
  \" = double quote
  \\ = backslash

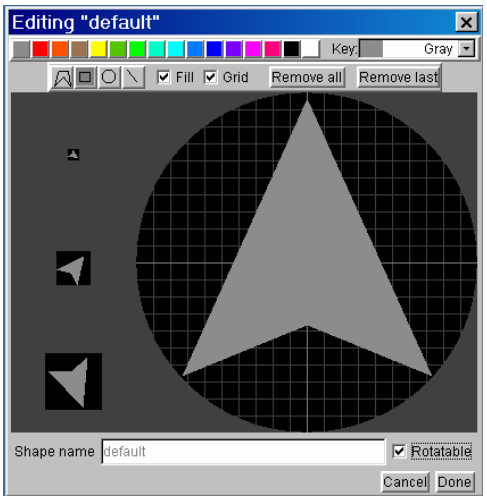
Scripting

- Turtle shapes are vector shapes. They are built up from basic geometric shapes; squares, circles, and lines, rather than a grid of pixels.
- Vector shapes are fully **scalable** and **rotatable**.
- A turtle's shape is stored in its **shape** variable and can be set using the set command.
- The **set-default-shape** primitive is useful for changing the default turtle shape to a different shape, or having a different default turtle shape for each breed of turtle.
- Use the Shapes Editor to create your own turtle shapes.

Shapes Editor

Use the Shapes Editor to create your own turtle shapes





References

In the Netlogo site you can find:

- the last version of Netlogo
- the Netlogo User Manual
- the new Netlogo model
- and a group-discussion about Netlogo

http://www.ccl.sesp.northwestern.edu/netlogo/